

Specification Basics

Abstraction for non-primitive data types (aka String Theory)

School of Computing
Clemson University
Clemson, SC 29634 USA

CPSC 2150: Wednesday, Mar. 15, 2023

Slide Sources:

- 1 Previous/Current CPSC 2150 Instructors
- 2 The lecture slides have also benefitted from instructors at Ohio State: Paolo Bucci, Wayne Heym, Paul Sivilotti, and Bruce Weide.

Last Class

- 1 Recall
 - ▶ Testing
 - ▶ Central Limitation of Testing
 - ▶ Tracing and Debugging
- 2 Formal Reasoning
 - ▶ Contracts
 - ▶ Complications
- 3 Tracing Table Examples
 - ▶ In-Class Exercises

Homework Assignment

- 1 Programming assignment 3 is posted on your lab Canvas page.

Lab This Week

Lab this week is attendance required! If you don't attend, you will not receive any credit for this week's lab.

- ▶ Talk to your course instructor if you have any concerns

Recap: Verification vs. Testing

- ▶ The goal of testing is to find failures
 - ▶ Lets us know that a fault exists
 - ▶ Can't use testing to show that no faults exist
- ▶ The goal of verification is to prove that no faults exist
- ▶ If formal reasoning/verification shows that our code works on all inputs, while testing does not, why don't we just verify everything and skip testing
 - ▶ Verification is very difficult and time-consuming
 - ▶ Requires a ton of formal mathematical logic
 - ▶ Proofs must be very rigorous
 - ▶ Not everything can be verified (right now)

Primitive Types and Abstraction

- ▶ If our parameters are primitive types like: `int` and `double`, then the math to use is obvious
- ▶ Though these types are represented internally with bits and bytes, users can view them *abstractly* like math variables
 - ▶ **Important Caveat:** Values are bounded in computing

Abstraction for Non-Primitives

- ▶ How do you specify generic lists?
 - ▶ Or stacks?
 - ▶ Or queues?

Example: Stack

- ▶ A **Stack** allows entries to be inserted and removed in **LIFO** (last-in-first-out) order

How to Specify This Interface?¹

```
public interface IntegerStack {  
  
    void push(Integer x);  
    Integer pop();  
    int depth();  
    ...  
  
}
```

¹Lecture 6

How to Specify This Interface?²

```
/**  
  Stack is abstractly a string of integers.  
  
  Initialization ensures:  
    Stack contains nothing, i.e., an empty string <>.  
  
  Constraints: length of a self <= MAX_DEPTH  
  */  
public interface IntegerStack {  
    public static final int MAX_DEPTH = ...;  
    ...  
}
```

²Lecture 6

How to Specify This Interface?³

```
public interface IntegerStack {  
    ...  
  
    /**  
        @pre    length of self, |self| < MAX_DEPTH  
        @post   self = x added to the front of #self  
    */  
    void push(Integer x);  
  
    /**  
        @pre    length of self, |self| > 0  
        @post   self = pop() removed from the front of #self  
    */  
    Integer pop();  
  
    ...  
}
```

³Lecture 6

Non-Primitive Types in General?⁴

- ▶ No stacks or queues in mathematics
- ▶ There are no “obvious” mathematical abstractions for new objects we will define and use
- ▶ Can we define a mathematical abstraction for stacks and queues?

⁴Lecture 6

Dual Ideas of Specification⁵

- ▶ Information hiding
 - ▶ Hide details unnecessary to use the `interface`
- ▶ Abstraction
 - ▶ Provide a “cover story” or explanation in user-oriented terms so they can understand the `interface`
- ▶ If there's not a formal mathematical notation to use, then we are stuck with an informal contract
- ▶ Informal $\neq$ Easy to Write

Specifications⁶

- ▶ Straightforward descriptions
 - ▶ **Push** pushes an object on a stack
 - ▶ How much do they help?
 - ▶ Carefully write the contracts. Rely on the interface description and invariants
- ▶ Use of metaphors
 - ▶ A **Queue** is like a line at a fast food restaurant
 - ▶ Do they generalize?
- ▶ Use of private implementation details
 - ▶ Remember information hiding!
 - ▶ Users shouldn't have to depend on implementation details

Specifications

- ▶ Can we think of stacks as mathematical sets abstractly, so a stack of integers would be a set of integers, and so on?
 - ▶ Can we use set theory notation?
 - ▶ Union
 - ▶ Intersection

Specifications

- ▶ Can we think of stacks as mathematical sets abstractly, so a stack of integers would be a set of integers, and so on?
 - ▶ Can we use set theory notation?
 - ▶ Union
 - ▶ Intersection
 - ▶ No!
 - ▶ If you push 3 and then 4, the stack is different from if you push 4 and then 3.
 - ▶ Sets have no order.
 - ▶ Whichever order you insert items, you get the same set $\{3, 4\}$
- ▶ You have to pick a proper abstraction!

So We Need to Digress Briefly

- ▶ We need a mathematical structure with order, so it is suitable as an abstraction of stacks, queues, etc.
- ▶ This structure is called **(mathematical) strings**

String Theory

- ▶ A string can be thought of as a series of zero or more **entries** of *any* other mathematical type, say, T
 - ▶ We will call this $\text{Str}(T)$
 - ▶ Also denoted in computing foundations as T^*

String vs. Str

- ▶ `String` is a programming type in Java
- ▶ `Str(T)` is a mathematical structure, like a set
- ▶ Strings in Java and C++ are a data type for the mathematical concept of *string of characters*
 - ▶ This is where `String` gets its name from though...

Math Notation for Strings

- ▶ The following notations are used when we write mathematics (e.g., in contract specifications) involving strings
- ▶ Notice two important features of strings:
 - 1 Unlike sets, there may be **duplicate** entries (in fact, there may be arbitrarily many of a given entry value)
 - 2 Unlike sets, the **order** of the entries is important

Math Notation for Strings

Empty String

- ▶ The *empty string*, a string with no entries at all, is denoted by `<>` or by `empty_string`

Math Notation for Strings

Denoting a Specific String

- ▶ A particular string can be described by listing its entries between $\langle$ and $\rangle$ separated by commas
- ▶ **Examples:**
 - ▶ $\langle 1, 2, 3, 2 \rangle$ ⁷
 - ▶ $\langle 'G', 'o' \rangle$ ^{8 9}
 - ▶ $\langle \rangle$ ¹⁰

⁷A *string of integer* value, whose entries are *integer* values: 1, 2, 3, 2.

⁸A *string of character* value, whose entries are *character* values: 'G', 'o'.

⁹We may also use the special notation: `'Go'` for the same *string of character* value.

¹⁰Now it can be seen that this notation: `empty_string` is a special case of the string literal notation.

Math Notation for Strings

Concatenation

- ▶ The *concatenation* of strings s and t , a string consisting of the entries of s followed by the entries of t , is denoted by $s \circ t$ ¹¹
- ▶ **Examples:**
 - ▶ $\langle 1, 2 \rangle \circ \langle 3, 2 \rangle = \langle 1, 2, 3, 2 \rangle$
 - ▶ $\langle \rangle \circ \langle 5, 2, 13 \rangle = \langle 5, 2, 13 \rangle$
 - ▶ $\langle \rangle \circ \langle \rangle = \langle \rangle$

¹¹In programming, Java uses `+` to concatenate two `String` values

Math Notation for Strings

Length

- ▶ The *length* of a string s , i.e., the number of entries in s , is denoted by $|s|$
- ▶ **Examples:**
 - ▶ $|<1, 2, 3, 2>| = 4$
 - ▶ $|<1, 2> \circ <3, 2>| = 4$
 - ▶ $|<'G', 'o'>| = 2$
 - ▶ $|<>| = 0$

Some More String Theory...

Reverse Function

- ▶ `Reverse(s)`
 - ▶ is the reverse order of string `s`
- ▶ **Examples:**
 - ▶ `Reverse(<1, 2, 2, 3>) = <3, 2, 2, 1>`
 - ▶ `Reverse(<>) = <>`

Some More String Theory...

Prt_Btwn Function

- ▶ `Prt_Btwn(m, n, s)` (Part-Between)
 - ▶ is the substring between positions `m` and `n` in the string `s`
 - ▶ includes `m`, up to `n`
- ▶ **Examples:**
 - ▶ `Prt_Btwn(0, 3, <7, 8, 9>) = <7, 8, 9>`
 - ▶ `Prt_Btwn(0, 1, <7, 8, 9>) = <7>`
 - ▶ `Prt_Btwn(1, 3, <7, 8, 9>) = <8, 9>`

Some More String Theory...

`Occurs_Ct` Function

- ▶ `Occurs_Ct(x, s)`
 - ▶ Is the number of times the item `x` occurs in the string `s`
- ▶ **Examples:**
 - ▶ `Occurs_Ct(2, <1, 2, 2, 3, 2>) = 3`
 - ▶ `Occurs_Ct(8, <1, 2, 2, 3, 2>) = 0`

Some More String Theory...

- ▶ More notations can be introduced and used when needed

Practice Exercises

- ▶ `Prt_Btwn(2, 4, <'a', 'b', 'c', 'd', 'e', 'f'>) = ?`

Practice Exercises

- ▶ `Prt_Btwn(2, 4, <'a', 'b', 'c', 'd', 'e', 'f'>) = ?`
- ▶ `Reverse(Prt_Btwn(1, 3, <7, 8, 9>)) = ?`

Practice Exercises

- ▶ `Prt_Btwn(2, 4, <'a', 'b', 'c', 'd', 'e', 'f'>)` = ?
- ▶ `Reverse(Prt_Btwn(1, 3, <7, 8, 9>))` = ?
- ▶ `|Prt_Btwn(1, 3, <7, 8, 9>)|` = ?

Practice Exercises

- ▶ `Prt_Btwn(2, 4, <'a', 'b', 'c', 'd', 'e', 'f'>)` = ?
- ▶ `Reverse(Prt_Btwn(1, 3, <7, 8, 9>))` = ?
- ▶ `|Prt_Btwn(1, 3, <7, 8, 9>)|` = ?
- ▶ `Occurs_Ct(3, (Prt_Btwn(1, 3, <3, 4, 3, 4, 3>))` = ?

Back to Stack Specification

- ▶ Now that we have a mathematical structure with order we can specify stacks!
- ▶ We'll cover this next time

Abstraction

```
/**  
 * mathematical model Str( $\mathbb{Z}$ )  
 * initialization ensures self = <>  
 */  
public interface IntegerStack {  
  
    ...  
  
    void push(Integer x);  
    Integer pop();  
    int depth();  
  
    ...  
}
```

- ▶ $\mathbb{Z}$ is the mathematical set of integers
- ▶ **initialization ensures** is the specification for the constructor.
 - ▶ Can also be written as: `self = empty_string`

Push Specification

```
/**
 * mathematical model Str(Z)
 * initialization ensures self = <>
 */
public interface IntegerStack {

    ...

    /**
     * @pre |self| < MAX_DEPTH
     * @post self = <x> o #self AND x = #x
     */
    void push(Integer x);

    ...
}
```

- ▶ `#self` is the value of this stack before `push`
- ▶ `x` is concatenated to the front of `#self` after `push`

Pop Specification

```
/**
 * mathematical model Str(Z)
 * initialization ensures self = <>
 */
public interface IntegerStack {

    ...

    /**
     * @pre |self| > 0
     * @post self = Prt_Btwn(1, |#self|, #self)
     *       <pop> = Prt_Btwn(0, 1, #self)
     */
    Integer pop();

    ...
}
```

- ▶ **self** is all of the items in the string after the first item
- ▶ **pop** is the first item in the string

Depth Specification

```
/**  
 * mathematical model Str(Z)  
 * initialization ensures self = <>  
 */  
public interface IntegerStack {  
  
    ...  
  
    /**  
     * @post depth = |self|  
     */  
    int depth();  
  
    ...  
}
```

- ▶ `depth` is just the length of the string

- ▶ Driver's Education and Formal Interface Specifications:
<https://www.cs.clemson.edu/resolve/research/reports/RSRG-11-05.pdf>

Homework Assignment

- 1 Programming assignment 3 is posted on your lab Canvas page.

Lab This Week

Lab this week is attendance required! If you don't attend, you will not receive any credit for this week's lab.

- ▶ Talk to your course instructor if you have any concerns

Summary

- 1 Recap: Verification vs. Testing
 - ▶ Primitive Types and Abstraction
- 2 String Theory
 - ▶ Empty String
 - ▶ Denoting a Specific String
 - ▶ Concatenation
 - ▶ Length
 - ▶ `Reverse` Function
 - ▶ `Prt_Btwn` Function
 - ▶ `Occurs_Ct` Function